1996

Software Component Search

Luqi

Software Component Search, with J. Goguen, D. Nguyen, J. Messeguer, D. Zhang, V. Berzins, Journal of Systems Integration (Special issue on Computer Aided Prototyping), Vol. 6, No. 1-2, pp. 93-134.
http://hdl.handle.net/10945/42343

Journal of Systems Integration, 6, 93–134 (1996)

Software Component Search

JOSEPH GOGUEN
University of California at San Diego, LaJolla, CA (on leave from Oxford University Computing Lab, Oxford, UK)

DOAN NGUYEN
McClellan Air Force Base, Sacramento, CA

JOSÉ MESEGUER
SRI International, Menlo Park, CA

LUQI
Naval Postgraduate School, Monterey, CA

DU ZHANG
California State University, Sacramento, CA

VALDIS BERZINS
Naval Postgraduate School, Monterey, CA

Abstract. An important problem in software development is to make better use of software libraries by improving the search and retrieval process, that is, by making it easier to find the few components you may want among the many you do not want. This paper suggests some ideas to improve this process: (1) Associate an *algebraic specification* with each software component; these specifications should include complete syntactic information, but need have only *partial* semantic information. (2) User queries consist of syntactic declarations plus results for *sample executions*. (3) User queries may be posed in standard programming notation, which is then automatically translated into algebraic notation. (4) Search is organized as *ranked multi-level filtering*, where each level yields a *ranked set of partial matches*. (5) Early stages of filtering narrow the search space by using computationally simple procedures, such as checking that the number of types is adequate. (6) Middle levels may find *partial signature matches*. (7) Pre-computed *catalogues* (i.e., indexes) can speed up early and middle level filtering. (8) Semantic information is used in a final filter with *term rewriting*, but complete verification is not attempted. (9) The series of filters is implemented *incrementally*, so as to backtrack to lower ranked components in case of failure. This approach avoids the need for complex theorem proving, and does not require any knowledge of algebraic specification from the user. Moreover, it does not require either specifications or queries to be complete or even fully correct, because it yields partial matches ranked by how well they fit the query. The paper concludes with a description of some preliminary experiments and some suggestions for further experiments.

Keywords: Component search, retrieval, software library, reuse, computer aided prototyping.

1. Introduction

Each year billions of dollars are spent on computer software. Much of this effort is spent on creating and testing new source code. In order to save money, increase productivity, and improve reliability, the Department of Defense is constructing repositories of reusable software components that can be used across applications. Devising an effective way to retrieve components from software libraries, referred to as the software component search problem (or simply, the *search problem*), is of increasing importance for many applications.

For example, rapid prototyping has been used to validate and refine system requirements, and to check the consistency of proposed designs, before undertaking a full implementation. Automated retrieval of relevant reusable software components is important for this area.

In practice, there may be no component in the software base that does exactly what is wanted, but there may be some component(s) that can be easily modified to do the job. This implies that given a query, we do not just seek components that match it exactly, but instead we seek a set of approximate candidates, ordered by how well they match the query. In other words, the choice set should consist of *ranked partial matches*.

These considerations motivate the following requirements for solutions to the search problem:

1. The retrieval process should be *automated*, since a software library may contain thousands of components, so that it would be virtually impossible for a human being to identify the desired component(s) quickly and accurately.

2. The retrieval process should be *accurate*, in the sense that the choice set should include the closest match, if there is one.

3. The search process should be *effective*, that is, it should be fast, and the choice set should not be too large.

4. The user interface should allow *flexible, easy* query formulation, and should provide helpful *feedback* to the user.

Luqi [19, 22] has suggested associating a semantic specification with each module in the software base to support retrieval against semantic queries, as has Goguen [7]. This idea has been shown viable in work reported in [21, 32, 33], where the algebraic specification language OBJ3 [6, 10, 16] was used in software search experiments in the context of the Computer Aided Prototyping System (CAPS) project (see Section 2.5). Recent work [14] has carried this further by developing a rigorous mathematical model, showing how to treat generic modules, how to use semantic information in a limited efficient way, and how to rank candidate modules by their likelihood of success (an earlier ranking method is described in [33]). Ranking modules by how well they satisfy the query makes the search process more *robust*, that is, better able to tolerate errors in the query and in how components are classified. This is useful because we must expect such errors in practice.

Given a query Q and a component M with corresponding specification T_M, then M is a *correct answer* for the query Q if there is a translation of the syntax of Q into the syntax of T_M such that each translated equation from Q is a consequence[1] of T_M. Finding a correct answer in this sense is really a theorem proving task that could take too much time to be practical if not limited. However, finding candidates that satisfy adequate necessary conditions for being a correct answer is a practical goal. This will allow many irrelevant candidates to be rejected, resulting in a more focused search and raising the confidence in the components found.

A brief summary of related work is given in Section 2. An overview of our software architecture for automated component retrieval is given in Section 3. Background information on algebraic specification, including basic definitions, is given in Appendix A. Sections 4 and

5 describe syntactic and semantic filtering, respectively. Some advanced topics in semantic filtering are discussed in Appendix B. An example illustrating the search process is given in Section 6, while Section 7 gives some concluding remarks and directions for further work. Many parts of this paper are adapted from [14].

For ease of reference, definitions, theorems, examples and algorithms are numbered sequentially on the same counter; thus, Definition 2 follows Example 1, and there are no Definition 1 or Example 2. Figures are numbered sequentially on a separate counter.

2. Background and Related Work

Previous work on reusable software component retrieval can be classified as classical, facet, AI, or pure specification. More information may be found in [5, 4, 17, 19, 25, 29]. We do not attempt to survey the entire relevant literature here, but instead we describe some publications that seem most closely related to the present work.

2.1. *Classical Approaches*

The most classical approach to retrieval is to classify items by keywords, and then search for items that have certain given keywords [23]. Experience shows that this works poorly for retrieving software components from even moderately large libraries. One problem is that the user must be familiar with both the classification scheme and the particular library. Also, it is very difficult to get both high precision and high recall[2]. In general, using a larger number of keywords raises precision at the expense of recall. This suggests that for ranked filtering, it would be most appropriate to use a small number of keywords.

Another classical approach is browsing. Browsing systems depend on links among the items to be searched, and upon the user following those links to find the desired item. Experience shows that browsing through large structures can be very frustrating and time-consuming. Often, existing links seem random or even perverse, while the links you really want may not be present.

2.2. *The Facet Approach*

Prieto-Diaz [30] has proposed using *facets*, which are groups of related terms in a subject area. For example, a facet to describe the functions performed by components might use terms chosen from *find, compare, sort, update, send, receive,* .... A scheme is developed in [30] to describe Unix components using four facets: the function performed by the component, the objects that are manipulated, the data structure used, and the system to which the function belongs. This provides a better description of Unix components than a pure keyword approach. However, it still relies on an informal description of components, using a limited set of facets and terms. Retrieval based on component behavior is lacking.

2.3. *AI Approaches*

AI-based work includes [3, 27], and some recent work by Henniger [18], which uses a knowledge-base and statistical information to retrieve reusable components, based on keyword search from texts describing the components. However, because the characterization of the component behavior is completely informal, the behavior is unpredictable.

2.4. *Specification-based Approaches*

Recent work using semantics for software component retrieval is reported in [25]. The primary aim is to check that retrieved components yield the behavior specified in the user's query, therefore increasing the precision of retrieval. Using formal specifications as search keys has two main problems. The first problem is practical: not all users are sophisticated enough to write formal specifications, much less correct ones. The second problem is that semantic matching is very time consuming, because some form of theorem proving must be done.

The Venari[3] project at Carnegie Mellon University, headed by Prof. Jeannette Wing, is devoted to retrieving components from software libraries, and has produced a number of interesting publications. Here we will not discuss their work on transactions and other infrastructural support for retrieval, but only their work on the search problem.

Rollins and Wing [31] discuss signature matching for retrieving higher-order functions from an ML library[4], using λProlog for matching user queries to component signatures. λProlog is used to implement matching modulo various theories, in order to support (what we call) partial matches[5]. They also use λProlog to check simple pre- and post- conditions for ML functions. Although this paper demonstrates that higher-order logic is useful for such applications, we feel that higher-order logic is more powerful and expressive than necessary, and that higher-order logic tools like λProlog are too inefficient. Of course, a higher-order language like ML requires the use of higher-order types, but these are first-order expressions, so that first-order matching could be used. Rollins and Wing point out that equational reasoning could dramatically increase precision, and they also discuss the possibility of specification matching.

Zaremski and Wing [34] extend this work. First, they consider signatures in two different senses, as the rank of a function, and as the interface of a module; the second sense involves search and retrieval of modules, not just of functions. Second, they consider a wider variety of matching procedures and their combinations, although some of these are needed only because of the awkwardness of the higher-order encoding of operation ranks (e.g., uncurrying). Third, they implemented their matching procedures in ML, experimented with retrieving functions from actual ML libraries, and presented some interesting statistics on these experiments.

In more recent work, Zaremski and Wing [35] focus on specification matching, using the Larch/ML interface language to express pre- and post-conditions in first order logic, and the Larch prover to verify that candidate components satisfy these conditions. Various senses of matching are defined, but neither ranking nor partial semantic matching are considered.

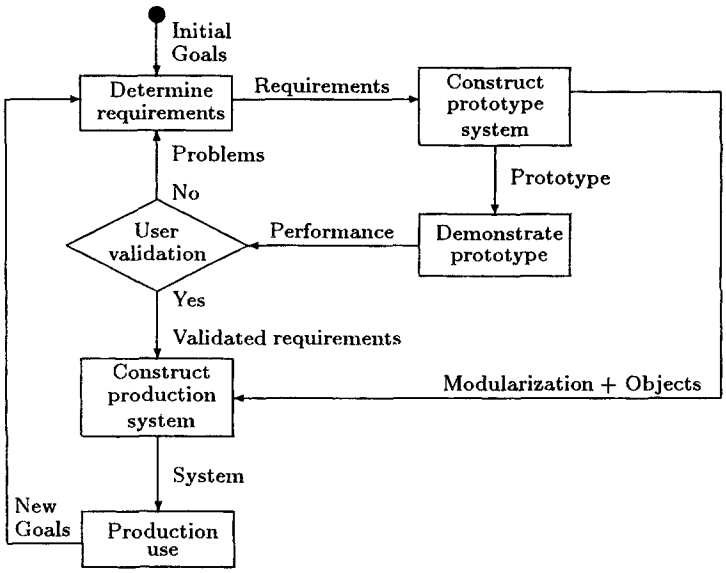

Figure 1. A prototyping lifecycle.

2.5. *The CAPS Approach*

The CAPS project at the Naval Postgraduate School, headed by Profs. Luqi and Valdis Berzins, supports rapid prototyping for hard real time embedded systems. CAPS consists of an integrated set of software tools that help design, translate and execute prototypes. These tools include an execution support system, a syntax directed graphical editor, an evolution control system, a change merge facility, automatic generators for schedule and control code, and facilities to support retrieving reusable components from a software base. The execution support system includes dynamic and static schedulers, a translator, and a debugger.

PSDL is the Prototyping Description Language of CAPS [20]; it is used to specify both prototypes and production software, and has a data flow like semantics. PSDL programs have two kinds of object, corresponding to abstract data types and abstract state machines; they localize the information for analyzing, executing and reusing independent objects. Executable Ada modules can be associated to atomic PSDL objects, and then CAPS can automatically generate "glue" code that composes these modules into a system having the structure described by PSDL. This generated code includes a schedule and tests for all real time constraints that have been declared. The system can then be compiled, executed, and tested. Error messages are produced during execution if constraints are violated. Figure 1 illustrates a prototyping lifecycle. It shows two places where component search can be used in such a lifecycle: in constructing a prototype system, and in constructing a production system.

The remainder of this subsection concentrates on work done in the CAPS project on

retrieving software components. The use of specifications in retrieving software components was suggested by Luqi [19]. This suggestion was refined in later work, including [32] and Steigerwald's PhD thesis [33]. In [33] it is assumed that each component has a fully expanded[6] algebraic specification written in OBJ [6, 10, 16], and that the user's query is also a fully expanded algebraic specification that the desired component should satisfy. A Prolog program was written using symbolic representations of signatures to find syntactic matches between the signature of the query and the signatures of components. For each match found, a semantic validation was done by evaluating patterns that represent the functions in the signature, first in the query specification[7], and then (after translation) in the specifications of the matched components; the results of these two evaluations are compared to determine the quality of each match. The approach developed in this series of papers is the inspiration for the approach taken in the present paper.

The system described in [33] has certain technical limitations. Its semantic basis is not well developed. Also, evaluating patterns with variables gives limited information about the semantic satisfaction of a syntactic match. In addition, since patterns can involve variables that may or may not be eliminated by rewriting, depending on syntactic peculiarities of the equations, it seems possible to have semantically equivalent specifications for which pattern evaluation would give conflicting answers, so that the match in question will appear not to satisfy its semantic requirements even though it really does. In addition, the approach is limited to total syntactic matches and to unparameterized components.

Ozdemir's master's thesis [28] describes a component retrieval system for the CAPS software base that uses keyword search and a browser. Both of these use PSDL for queries and for components. [28] also provides a graphical user interface and facilities for integrating retrieved components into prototype systems, including techniques for transforming retrieved modules. A better developed version of these ideas appears in the masters thesis of Dolgoff [2]. This work eludes retrieval of generic modules and handles subsort matching.

2.6. Discussion

In comparing our approach with others, the following points may be noted: (1) Our approach focuses on comparing formal specifications of components using ground equation test cases as queries. However, our theoretical work shows how to automatically generate suitable ground equations from non-ground equations in the query. (2) Users do not need to deal with formal specification notation, but instead can express queries in a standard programming notation, which is automatically translated into algebraic notation. (3) We seek to achieve both efficiency and effectiveness by imposing a series of increasingly stringent filters that use both syntactic and *partial* semantic information about components. (4) A rank is provided on components in the choice set, measuring how well they fit the user's query. (5) We allow generic modules in the software base. (6) Our approach not only focuses on the problem of retrieving components, but also deals with structuring the software base to facilitate search. (7) Users can give *selection criteria* to control the search and display of retrieved components. (8) Besides returning the ranked components, we also report information to help the user reformulate the query in case no suitable component was found. (See Figure 2.)

3. Architecture of the Search Process

This paper proposes an approach to the automated retrieval of reusable software components from a software base, continuing work reported in [19, 21, 32, 33, 7] and especially [14]. The approach is based on the following assumptions: (1) the components in the software base are written in a modern programming language, e.g., Ada, that has strong typing, can package together a number of operations over common data representations, and allows generic modules having a number of parameter types and operations; (2) each component has an algebraic specification[8] with equations that are Church-Rosser and terminating[9]; and (3) the user's query is a partial algebraic specification, typically consisting of a signature and some ground equations. Assumption (2) is not really limiting, because specifications need not completely characterize the behavior of components, and simple partial specifications are usually Church-Rosser and terminating[10]. Similarly, assumption (3) is not limiting, because there is no need for users to be familiar with algebraic specification: query signatures can be expressed as declarations in some familiar programming or specification notation, and the query can be described in terms of the results of executing simple programs. Note that the software base may contain generic modules and queries may seek to identify a generic module having certain semantic properties.

In our approach, search is organized as a series of increasingly stringent filters on candidate components. We first filter components by comparing their signatures with that of the query. This is accomplished by *signature matching*, which looks for maps that translate the type and function symbols of the query into corresponding type and function symbols of candidate components. A first stage of signature filtering can compare pre-computed *syntactic profiles* of components with the profile of the query. These profiles are special data structures that support an efficient approximation of signature matching. Signature matches can be *partial*, in that only part of the functionality the user seeks may actually be available. Traditional search methods, such as keyword search, could also be used as early filters, if the appropriate information is available[11]. Profile matching should be followed by full signature matching.

After this, semantic filters rank components by how well they satisfy the equations in the query. In this process, equations that are logical consequences of the query specification are translated through the signature matches into equations whose proof is attempted in the candidate specifications. For greatest efficiency, it is desirable to restrict queries to be ground equations; these correspond to simple straight line programs. Techniques are described in Appendix Sections B.1 and B.2 for generating ground equations from non-ground queries. The candidates in the choice set are ranked according to their likelihood of success. If the closest match is partial, the user will need to modify the closest matching component. This whole process can be made iterative.

Our present knowledge is not sufficient to say that any particular sequence of filters is the most appropriate. However, it does seem clear that simple filters should be applied first, in order to eliminate as many components as possible with the lowest possible cost. Therefore syntactic filtering, and possibly keyword filtering, should come before any semantic filtering is attempted. It is also clear that pre-computed catalogues (i.e., indexes) should be used, instead of pulling all the components out of the library for each search. Figure 2 shows one

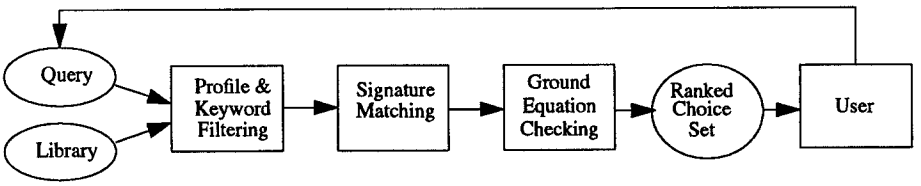

Figure 2. One organization of multi-level filtering.

possible multi-level filtering architecture; the top line is to indicate user modification of the query in light of the final filtering results.

4. Syntactic Filtering

Syntatic filtering uses non-behavioral information about components, such as keywords and interface declarations. Our approach involves three levels of syntatic filtering. In the first, profile filtering computes indexes which partition the software library in a way that speeds up signature matching. Then keyword filtering further reduces the choice set. And finally, signature matching finds the maps that translate the type and function symbols of the query into the corresponding type and function symbols of the candidate components.

4.1. Keyword Matching

Despite its weaknesses, keyword search is still useful, because it is easy to use, inexpensive to implement, and good for indexing components. However, we use keyword filtering cautiously, with a limited number of general keywords that controlled by a system administrator. Keywords describe categories of components and their relationships to the other components. Sample categories might be data structures, mathematical functions, sort/search routines, and navigation functions.

We use the following rank function to measure how close the keywords of a query, Kw_Q, are to the keywords of a component, Kw_M:

$$KeywordRank(Kw_Q, Kw_M) = |Kw_Q \cap Kw_M|/|Kw_Q| \ .$$

Note that this function measures recall.

4.2. Signature Matching

This subsection introduces and illustrates the basic concepts of signature matching, under the assumption that there are no subsort relations; the more complex situation when S has subsorts (i.e., a partial order relation $\leq$) is discussed in Sections 4.3 and 4.4 below. We assume[12] that each component M in the library has an associated algebraic specification

T_M of the form (S, Σ, E), where (S, Σ) is a signature with a set S of sorts and a set Σ of functions whose arguments and results have sorts in S, and where E is a set of equations stating properties that the functions in Σ should satisfy. We also assume that queries are algebraic specifications, of the form (R, Ω, E'). The following illustrates these notions, using the notation of OBJ3; precise definitions for signature, specification, etc. can be found in Appendix A.

Example 1. The algebraic specification for a list of identifiers module in our library might have a sort set S containing the sorts `Id`, `List`, and `Bool`, and function symbols for the empty list, denoted `null`, an append operation `*`, and a function to test whether an element is in the list, with the following syntax:

```
sorts Id List Bool .
op nil : -> List .
op _*_ : Id List -> List .
op _in_ : Id List -> Bool .
```

The equations in the a specification might be:

```
I in nil = false .
I in (I' * L) = if (I == I') then true else I in L fi .
```

(Note that the "underscore" characters "_" are place holders, indicating where the arguments should go in "mixfix" syntax for functions.)

We also assume: that the library has a set of **basic sorts**[13] for commonly used types like Booleans, identifiers, integers, and floating point numbers; that the names of these basic types and their associated basic operations are identical in the specifications and in the code; and that the library modules and the algebraic specifications for the basic types also have the same names. The following two definitions are from [14]:

Definition 2. Given two signatures (S, Σ) and (S', Σ'), a **permutative signature map** V: $(S, \Sigma) \to (S', \Sigma')$ consists of injective functions V: $S \to S'$ and V: $\Sigma \to \Sigma'$ such that for each function symbol f: $s_1 \ldots s_n \to s$ in Σ, there is a permutation π such that $V(f)$: $V(s_{\pi(1)}) \ldots V(s_{\pi(n)}) \to V(s)$ is a function symbol in Σ'. A **partial signature match** V: $(S, \Sigma) \to (S', \Sigma')$ is a permutative signature map V: $(S_0, \Sigma_0) \to (S', \Sigma')$ where (S_0, Σ_0) is a subsignature of (S, Σ); it is **total** if $(S_0, \Sigma_0) = (S, \Sigma)$.

The assumption that V is injective on both sorts and operations is reasonable, because otherwise the user would be asking for two or more things that are not actually different.

Definition 3. Given a library and a query $Q = (R, \Omega, E')$, the **signature choice set** for Q consists of all signature matches V: $(R, \Omega) \to (S, \Sigma)$ where $T_M = (S, \Sigma, E)$ is the specification of some component M, and where each match V is the identity on the set of basic types and their basic function symbols. To simplify notation and make explicit the module specification associated with a signature match, we may write V: $Q \to T_M$ for a

signature match V: $(R, \Omega) \rightarrow (S, \Sigma)$ such that $T_M = (S, \Sigma, E)$ is the specification of a component M.

The more complex situation when S has subsorts (i.e., a partial order relation $\leq$) is discussed in Sections 4.3 and 4.4 below. Note that the semantic information in the equations is ignored in syntactic matching.

Example 4. Assume that a user wants to find a module for sets of identifiers, where the sort `Id` of identifiers and the sort `Bool` of Booleans are basic sorts. Suppose that the signature for such a query includes an empty set, functions for adding and deleting an identifier from a set, and a function to test whether a given identifier is in the set. This can be put in OBJ3 notation as follows:

```
sorts Id Bool Set .
op null : -> Set .
op _+_ : Set Id -> Set .
op _-_ : Set Id -> Set .
op _in_ : Id Set -> Bool .
```

Suppose that the library, among other things, contains a list of identifiers module whose specification has the signature shown in Example 1. The set of all signature matches from the query to this module has 17 elements. The least defined element V_1 is the identity map on `Id` and `Bool`, and is undefined elsewhere. V_2 extends V_1 by mapping `Set` to `List`. The maps V_3–V_6 each extend V_2 by respectively sending `null` to `nil`, sending `_+_` to `_*_`, sending `_-_` to `_*_`, and sending `_in_` to `_in_`. The rest are obtained as unions of V_3–V_6 satisfying the requirement of being injective, as follows:

- $V_7 = V_3 \cup V_4$;
- $V_8 = V_3 \cup V_5$;
- $V_9 = V_3 \cup V_6$;
- $V_{10} = V_4 \cup V_6$;
- $V_{11} = V_5 \cup V_6$;
- $V_{12} = V_3 \cup V_4 \cup V_6$;
- $V_{13} = V_3 \cup V_5 \cup V_6$.

Our intuitive knowledge of the behavior of sets and lists tells us that the best possible match is V_{12}.

4.3. *Profile Matching*

The computations for signature matching would be very expensive if it were necessary to try all possible ways of matching the functions and sorts of queries with those of components.

It is therefore highly desirable to cut down the search space. This can be done. For example, if a query has a function f: $AAB \rightarrow B$ and a component has a function g: $ABC \rightarrow D$, then it is obvious that these functions cannot match, because their arguments have different sort patterns; there is no need to compute all possible sort maps to draw this conclusion.

The purpose of profile matching is to speed up signature matching. Profile matching is actually an efficient approximation of signature matching. A profile is a sequence of numbers that describes how the sorts associated with an operation are organized. We can quickly determine whether two given operations could possibly match by comparing their profiles, and hence quickly identify query operations and test cases that will necessarily fail. Profile matching uses pre-computed profile indexes to partition the library, and profile bags are used as search keys in seeking components having suitable signatures.

We now introduce some concepts to help us define profiles. The **sort groups** of an operation are bags (i.e., multisets) consisting of two or more sort occurrences from the rank (i.e., the argument plus value sorts) of the operation that are related under the relation $\equiv$, which is the transitive-symmetric closure of the ordering $\leq$ on sorts. The **unrelated sort group** is the bag (which is actually a set) of all sort occurrences that are not in any sort group.

Definition 5. The **profile** of an operation is a sequence of integers, defined as follows:

1. The first integer is the total number of occurrences of sorts.
2. If the total number of sort groups, N, is greater than 0, then the second to $(1 + N)^{th}$ integers are the cardinalities of the sort groups, in descending order.
3. The $(2 + N)^{th}$ integer is the cardinality of the unrelated sort group.
4. The $(3 + N)^{th}$ integer is:

   0 if the value sort is different from any of the argument sorts; and

   1 if the value sort belongs to some sort group.

A signature map can relate two operations only if they have the same profile (i.e., the same number of sort occurrences, the same number of sort groups, the same sort group cardinalities, and the same unrelated sort group cardinality).

Example 6. Some sample profiles are shown in Table 1, where A, A', B, C, E, F, G are sorts, and A' is a subsort of A.

4.3.1. Software Base Partitioning with Profiles

We assume that each component C in the software base $\mathcal{L}$ has an implementation part and a specification part. The implementation part is implementation language source code, while the specification part is written in the CAPS prototyping language PSDL and/or OBJ3.

Table 1. Some operations and their profiles.

Operation	Profile
$\rightarrow A$	110
$EF \rightarrow G$	330
$AA \rightarrow B$	3210
$ABBCA \rightarrow C$	622201
$CCBAA \rightarrow B$	622201
$CCBAA \rightarrow A$	63211
$CCBAAA' \rightarrow A$	724211

For each component C, let $b(C)$ denote the bag of all profiles that occur in the signature of C; this information may be extracted either from the source code for C, or from the specification of C, and is called the **profile of** C. If b is a bag, let $|b|$ denote the total number of occurrences of items in b.

If β is a bag of profiles, let $P(\beta)$ denote the set of all components C in the software base such that the profile of C is β, i.e., let

$$P(\beta) = \{C \in \mathcal{L} \mid \beta = b(C)\} .$$

Let P be the bag of all profiles that occur in the software base, and let Π denote the set of all subbags β of P such that $P(\beta)$ is non-empty. Then Π is a partially ordered set under bag inclusion, and it induces a partition of the software base. Π can be represented graphically as a so-called **Hasse diagram**, having as its nodes the elements of Π, with an edge upward from β_1 to β_2 iff $\beta_1 \subset \beta_2$ and there is no node β_3 such that $\beta_1 \subset \beta_3 \subset \beta_2$.

Given a profile p in P, we define the **frequency** of p to be the number of profiles β in Π that contain p, i.e.,

$$Freq(p) = |\{\beta \in \Pi \mid p \in \beta\}| .$$

Let us call bags β' such that $|\beta'| = 1$ **bottom nodes**, and define Π' to be Π plus all bottom nodes β' not already in Π, again ordered by bag inclusion. This is the structure actually used in the implementation. We call Π' a **software base partition**. Note that we can still talk about the frequency of profiles in Π', and that $Freq(p)$ can be computed by recursively following edges upward in the Hasse diagram, starting from the bottom node $\{p\}$.

Example 7. To illustrate the above concepts, assume a small software base such that:

$$b(C1) = b(C2) = b(C3) = \{p_1, p_2\} ,$$
$$b(C4) = b(C5) = \{p_1, p_2, p_3\} ,$$
$$b(C6) = b(C7) = b(C8) = \{p_1, p_2, p_2, p_4, p_6\} ,$$
$$b(C9) = b(C10) = \{p_4, p_5\} ,$$
$$b(C11) = b(C12) = b(C13) = \{p_1, p_2, p_3, p_4\} ,$$
$$b(C14) = b(C15) = \{p_1, p_2, p_4, p_5\} .$$

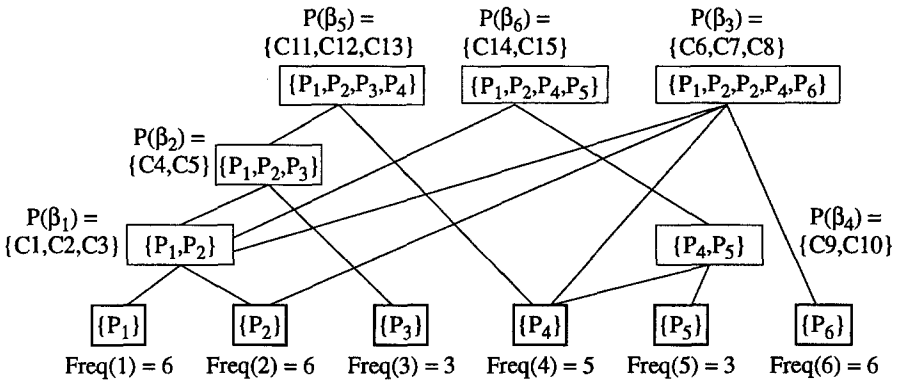

Figure 3. A software base partition.

Then

$$P = \{p_1, p_2, p_3, p_4, p_5, p_6\} ,$$

and if we assume

$$\beta_1 = \{p_1, p_2\},\ \beta_2 = \{p_1, p_2, p_3\},\ \beta_3 = \{p_1, p_2, p_2, p_4, p_6\}, \beta_4 = \{p_4, p_5\} ,$$
$$\beta_5 = \{p_1, p_2, p_3, p_4\},\ \beta_6 = \{p_1, p_2, p_4, p_5\} ,$$
$$\beta'_1 = \{p_1\},\ \beta'_2 = \{p_2\},\ \beta'_3 = \{p_3\},\ \beta'_4 = \{p_4\},\ \beta'_5 = \{p_5\},\ \beta'_6 = \{p_6\} ,$$

then we have that

$$\Pi = \{\beta_1, \beta_2, \beta_3, \beta_4, \beta_5, \beta_6\} ,$$
$$\Pi' = \{\beta'_1, \beta'_2, \beta'_3, \beta'_4, \beta'_5, \beta'_6, \beta_1, \beta_2, \beta_3, \beta_4, \beta_5, \beta_6\} ,$$

and also that

$$P(\beta_1) = \{C1, C2, C3\},\ \ P(\beta_2) = \{C4, C5\},\ \ P(\beta_3) = \{C6, C7, C8\},$$
$$P(\beta_4) = \{C9, C10\} ,\ P(\beta_5) = \{C11, C12, C13\},\ \ P(\beta_6) = \{C15, C16\} .$$

Let us also assume that

$$Keywords = \{A, B, C, D\} ,$$
$$Kw(C1) = \ldots = Kw(C5) = \{A, B\} ,$$
$$Kw(C6) = \ldots = Kw(C10) = \{B, C\} ,$$
$$Kw(C10) = \ldots = Kw(C15) = \{C, D\} .$$

Figure 3 shows the software base partition for this data.

In our implementation, each block of the partition is described as a set of pointers to indexes in the *ComponentLookupTable*, which has a cell for each component, containing

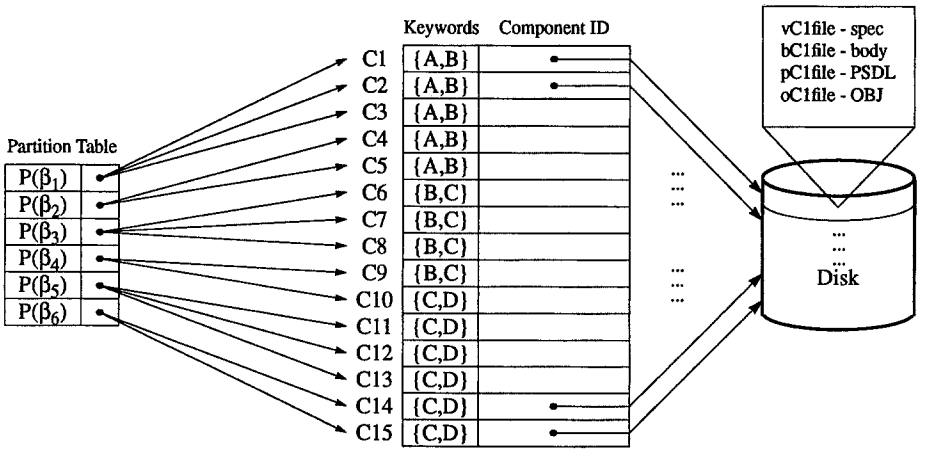

Figure 4. A software base index structure.

the keywords for the component, and a pointer to physical disk location of the component. These pointers are called *ComponentIDs*. Finally, the software base keeps a separate keyword table for storing keyword identifications (*KeywordIDs*), called the *KeywordTable*. Figure 4 shows the software base index structure for the data in Example 7.

We organize the physical file structure of a software base as follows:

1. A file containing a PSDL specification, for the translation and scheduling for a prototype using the component.

2. A file containing an OBJ3 specification, used for both signature and semantic matching.

3. A file containing an Ada specification, which is imported to become (what in CAPS is called) an atomic operator's Ada specification in a prototype, after some modifications.

4. A file containing an Ada body, corresponding to the semantic part. It is imported to a prototype, to become an atomic operator's Ada body, after some modifications.

This organization supports traditional graph search algorithms, such as depth first search, to implement DBMS operations such as initialize, delete, add, and retrieve. There could also be a file for compiled code, which could replace the use of OBJ3 for some aspects (see section B.4).

Given a query profile β_Q and component profile β_C, we define the function *ProfileRank* to measure how close the query profile is to the component profile as follows:

$$ProfileRank(\beta_Q, \beta_C) = |\beta_Q \cap \beta_C| / |\beta_Q| \,.$$

This is again a recall measure.

4.3.2. *Retrieving Components with Profile Matching*

Given a query Q and a software base with Hasse diagram G, let β_Q be the profile of the query; it is the bag of profiles of operations in the query. Then components are retrieved by depth first search in G for candidate components belonging to blocks that intersect with β_Q, by the following algorithms:

Algorithm 8 *FindCandidateComponents*

Input: β_Q, the query profile.
Kw_Q, the query keywords.
G, the Hasse diagram for the software base.
KeywordTable and *ComponentLookupTable*.
Output: *CandidateTable*, containing candidate components and their ranks, and invalid query operation(s) and test case(s).
(1) Find the frequency of profiles $p \in \beta_Q$, by recursive traversal of G from bottom nodes in β_Q.
If $Freq(p) = 0$, then mark p and its associate test case(s) invalid.
(2) For each keyword in Kw_Q find its ID value from the *KeywordTable*.
(3) Initialize v to be the empty bag.
(4) Call $DFSFW(G, v)$ to search for candidate components and rank them.
(5) Using the computed rank, sort the candidates in *CandidateTable*, according to the users selection criteria.
(6) Report invalid operation(s) and test case(s).

Algorithm 9 *DFSFW*

Input: G, a Hasse diagram.
v, a vertex in G.
CandidateTable, containing candidate components and their ranks.
Output: *CandidateTable*.
(1) Mark v visited.
(2) If any profile in β_Q is also in β_v then
Calculate the rank of profile matches between β_Q and β_v.
For each component pointer at $p(\beta_v)$ do
Index to an element in *ComponentLookupTable*.
Calculate the rank of keyword matches between Kw_Q and Kw_M.
Calculate the complete rank, *ProfileKeywordRank*,
by multiplying the keyword and profile ranks.
Store *ProfileKeywordRank* and *ComponentID* in *CandidateTable*.
(3) For each vertex n adjacent and above v do
If n not visited then $DFSFW(G, n)$.

4.4. *Signature Matching Algorithm*

The signature matching algorithms seek to find good partial signature maps V: $(S, \Sigma) \to (S', \Sigma')$, where (S, Σ) is the signature of the query Q and (S', Σ') is the signature of a

component M. Recall that the basic sorts are those common to all modules. The algorithms below take advantage of the following requirements for a signature map V, with sort map V_S: $S \to S'$ and operation map V_Σ: $\Sigma \to \Sigma'$:

1. V_S must be injective.
2. V_S must preserve the subsort relation, i.e., $s_1 \leq s_2$ in S implies $V(s_1) \leq V(s_2)$ in S'.
3. V_S must preserve basic sorts.
4. V_Σ must be injective.
5. V_Σ must preserve basic operations.
6. The profile of each operation f in Σ must be the same as the profile of $V_\Sigma(f)$ in Σ'.
7. If an operation f in Σ has argument sorts $s_1, \ldots, s_n$, and if $V_\Sigma(f)$ in Σ' has argument sorts $s'_1, \ldots, s'_n$, then there must be a permutation π of $\{1, \ldots, n\}$ such that $V(s_{\pi(i)}) \equiv s'_i$ for $i = 1, \ldots, n$, and such that $V(s) \equiv s'$.

This generalizes Definition 2 to the case where there are subsorts. Dolgoff [2] made some initial studies of signature matching with subsorts.

Given a signature match V, the measure of how close the signature of the query Q is to a component signature is given by the following:

$$\mathit{SignatureRank}(V, Q) = |V.\Sigma|/|Q.\Sigma|\,,$$

where $Q.\Sigma$ is the signature of the query and $V.\Sigma$ is the subsignature of $Q.\Sigma$ that is actually matched by V.

The following three algorithms compute the signature matches. The algorithm *SignatureMatch* computes V_Σ, *SortAssign* computes V_S, and *SortAssignable* determines whether a sort assigment can be made.

Algorithm 10 *SignatureMatch*

The query signature, (S, Σ).
The component signature, (S', Σ').
Output: The signature map, $V = (SAL, OAL)$.
Variables: *SAL* is the sort assignment list.
OAL is the operation symbol assignment List.
OVL is the operator visit List for storage of visited operators.
f_i, $f_{j'}$ are each an index to an operator in Σ, Σ'
Lopc is a list of flags indicating whether operations in Σ' are occupied or not.
θ_i is a state variable consisting of SAL, OAL, OVL, $Lopc$, f_i.
Stack is a FIFO stack for storing/retrieving θ_i.
m is an indexed map of V.
Loop-I: For f_i to $|\Sigma|$ do:
(1) Initialize: θ_1 with *SAL*, *OAL*, *OVL* $\leftarrow$ *Null*, *Lopc* $\leftarrow$ *False*.
Stack $\leftarrow$ *Null*.

(2) Find a $f_{j'}$ such that $Lopc_{f_{j'}} \neq True$ and $OVL_{f_{j'}} \neq f_i$.
Find $pf_{j'}$ such that $OVL_{pf_{j'}} = f_i$.
(3) If f_i = Null
If *Stack* is *Null* then
If $V \neq Null$ then stop.
Otherwise, return to Loop-I for next iteration.
Otherwise, $V_m \leftarrow (SAL, OAL)$, $m \leftarrow m + 1$, $\theta_i \leftarrow pop(Stack)$.
Return to step (2).
(4) Otherwise, if $pf_{j'} \neq Null$, then $Lopc_{pf_{j'}} \leftarrow False$.
$Lopc_{f_{j'}} \leftarrow True$ and $OVL_{f_{j'}} \leftarrow f_i$.
(5) If $profile(f_i) = profile(f_{j'})$ and there is sort assignment(s), then
$Stack \leftarrow push(\theta_i, Stack)$. $OAL \leftarrow (f_i$ symbol, $f_{j'}$ symbol).
For each possible sort assignment do:
Update *SAL*.
$Stack \leftarrow push(\theta_i, Stack)$
$\theta_i \leftarrow pop(Stack)$.
If $f_i \neq |\Sigma|$ then $f_i \leftarrow f_{i+1}$
Otherwise, $f_i \leftarrow Null$ and return to step (3).
(6) Return to step (2).

In discussing the algorithms below, we will use the following terminology: A non-basic sort is called **confined** if it is mapped from some sort under V, or is related to a confined sort under $\equiv$; a non-basic sort that is not confined is called **unconfined**. Two sorts s, s' that are not related under V are called **unrelated**, written $s \bowtie s'$. The algorithms below try to map unassigned sorts to appropriate values, and to assign appropriate values to the parameters of generic components.

Algorithm 11 *SortAssign*
Input: S_d is the argument sort set of a query operator.
S'_d is the argument sort set of a component operator.
s_r is the value sort of a query operator.
$s_{r'}$ is the value sort of a component operator.
SAL is current sort assignment list.
Output: *SAL'* is a set of possible sort assignments.
Variables: *TSAL* is temporary sort assignment list.
SVL is the sort visiting list, for storing pairs of indexes of element in S_d and S'_d.
d_i, d'_j are indexes to argument and value sort of S_d,S'_d.
$p_{j'}$ is the previous d_j index that is being occupied by d_i.
Lds is a list of flags indicating whether argument sorts in S'_d are occupied or not.
β_i is state variable consists of *TSAL*,*SVL*, d_i,*Lds*.
m is an index variable to SAL'.
(1) Initialize β_i with $TSAL \leftarrow SAL$; $SVL, Lds \leftarrow Null$; $d_i \leftarrow d_1 \in S$.
Initialize $m \leftarrow 1, Lds \leftarrow False, SAL \leftarrow Null$.
(2) If s_r is sort assignable to s'_r then $TSAL \leftarrow (s_r$ sort, s'_r sort).
Otherwise, return;

Table 2. Sort assignments.

$q_s \rightarrow c_s$	$q_s \bowtie c_s$	$q_s \equiv c_s$
Basic → *Confined*	F	T
Basic → *Unconfined*	T	NA
Basic → *Basic*	F	T
Confined → *Basic*	F	T
Confined → *Unconfined*	F	NA
Confined → *Confined*	F	T
Unconfined → *Basic*	T	NA
Unconfined → *Confined*	F	NA
Unconfined → *Unconfined*	T	NA

(3) Find a d'_j such that $Lds_{d'_j} \neq True$ and $SVL_{d_{j'}} \neq d_i$. Find a $p_{j'}$ such that $SVL_{p_{j'}} = d_i$.
(4) If $d'_j = Null$, then check the Stack.
If it is *Null*, then return.
Otherwise, $\beta_i \leftarrow pop(Stack)$. Return to step (3).
(5) Otherwise, if $p_{j'} \neq Null$, then $Lds_{p_{j'}} \leftarrow False$, $Lds_{d_{j'}} \leftarrow True$ and $SVL_{d'_j} \leftarrow d_i$.
(6) If d_i is sort assignable to d'_j, then $Stack \leftarrow push(\beta_i, Stack)$.
$TSAL \leftarrow (d_i$ sort, $d_{j'}$ sort). Assign $d_i \leftarrow d_i + 1$.
If $d_i > |S|$, then $SAL'_m \leftarrow TSAL$, $m \leftarrow m+1$ and $\beta_i \leftarrow pop(Stack)$.
(7) Return to step (3).

Algorithm 12 *SortAssignable*
Input: q_s is a query sort.
c_s is a component sort.
SAL is the current sort assignment list.
Output: *Aflag* is a flag to indicate whether an assignment can be made.
(1) Determine the sort relation between q_s and c_s by consulting a row of Table 2.
(2) Determine the sort type of q_s and c_s by consulting a column of Table 2.
(3) Look up the result in Table 2 and assign that value to *Aflag*.
(4) If *Aflag* is *True* then
If all subsorts of q_s and subsorts c_s are assignable by recursive calls of *SortAssignable*
Aflag is *True*
Else
Aflag is *False*
(5) Return *Aflag*.

5. Semantic Filtering

The choice set of signature matches for a query can be further narrowed by semantic filtering. Under certain very reasonable assumptions, this can be done efficiently by checking

satisfaction of certain semantic conditions that are necessary for any correct answer to the query.

We discuss only unparameterized specifications here, but intend to consider the extension to parameterized specifications in a future publication. We assume throughout that the specifications associated with components have equations that are Church-Rosser and terminating. This means we can apply the equations from left to right to simplify a term to a unique simplest possible form, called its canonical form, which can be regarded as the result of evaluating the term (see Theorem 14 in Appendix A). For example, the canonical form of the term

```
a in (b * (a * nil))
```

using the equations in Example 1 is `true`.

Our semantic validation procedure takes ground equations $t = t'$ from the query specification Q and tests them for satisfaction in the candidate specifications. Ground equations, that is, equations whose terms have no variables, are particularly useful here, because any ground equation provable from an equational theory Q is satisfied by the standard model of such a theory, i.e., the initial algebra T_Q of Q, so that those equations are also satisfied under the initial (standard) interpretation of Q.

This is important because either the query Q or the component specifications T_M may have an initial interpretation, so that proving the semantic correctness of a signature match $V\colon Q \to T_M$ could require complex theorem proving to check inductive consequences. The great advantage of ground equations is that their translations $V(t) = V(t')$ must be provable from T_M in order for V to be correct, regardless of whether Q and T_M are interpreted initially or loosely. Therefore, we can use them under either interpretation to further restrict the search for correct answers to the query Q. In addition, since T_M is assumed to be Church-Rosser and terminating, we can automatically settle the issue of whether $V(t) = V(t')$ is provable from T_M by comparing the normal forms of $V(t)$ and $V(t')$ after rewriting them with the equations in T_M (again, see Theorem 14 in Appendix A).

Appendix Sections B.1 and B.2 describe two techniques for automatically generating ground equations from a non-ground query Q, whose translations can be automatically checked for each signature match of a query. The first assumes that the query Q is also Church-Rosser and terminating, while the second doesn't. The resulting sets of ground equations provide two successive filters for semantic validation of signature matches for the query Q.

The next subsection explains how to rank members of the choice set based on semantic filtering.

5.1. Ordering Semantic Matches

Since many signature matches for a query might be found in a large library, it is important to narrow the search by using whatever semantic information is available at each filtering stage, either from the specification T_M, or as explained in Section B.4, the compiled component M, or both. For this, a sound way of ordering the matches according to their relative degree

of semantic correctness is needed. Following [14], we define below two measures that can be used for this purpose. These measures assign a value to each pair (V, I_V) consisting of a signature match for the query Q, and whatever information I_V is available at that stage about equations that have passed or failed the semantic checks. Such measures can be used independently or in combination to define choice sets.

Given a match V, the information I_V available for it may be either syntactic or semantic. Syntactic information will include the functions in the query's signature for which the match is defined and their translation under such a match. Semantic information will include the results of checking correctness of ground equations after translating them through V. For each such ground equation and match V three things can happen:

- the translated equation, after reducing each side to normal form using the equations in the specification of the module matched by V, yields an *identity*; therefore this equation has *succeeded* for this match;
- the translated equation, after reducing each side to normal form using the equations in the specification of the module matched by V, yields an equation whose two sides are different; therefore this equation has *failed* for this match;
- the equation could not be translated, because the match V was undefined for some of the functions appearing in the terms of the equation; this is also a kind of *failure*.

Note that we can associate to each equation a function symbol, namely the top function symbol of its left side. Therefore, we can assume that the semantic information in I_V is organized by function symbol so that for each such symbol f in the query's signature we have a set of ground equations for it, and information about their success or failure for the match V.

The first measure μ assigns to each pair (V, I_V) a real-valued function $\mu_{(V,I_V)}$: $\Omega \to \Re$, where $\Re$ is the real numbers and Ω is the signature of the query Q, defined as follows:

- If $V(f)$ is undefined, then $\mu_{(V,I_V)}(f) = 0$.
- $V(f)$ is defined but has no ground equations associated with it, then $\mu_{(V,I_V)}(f) = 1$.
- Otherwise, $\mu_{(V,I_V)} = 1 + \mathit{success}(f)/\mathit{equations}(f) - \mathit{failure}(f)/\mathit{equations}(f)$, where $\mathit{success}(f)$ is the number of successful checks for ground equations $t = t'$ with t having f as its top function symbol reported in I_V, $\mathit{equations}(f)$ is the total number of such equations, and $\mathit{failure}(f) = \mathit{equations}(f) - \mathit{success}(f)$; that is, we account as failures both failure in the translation and failure to evaluate to an identity after translation.

Consider now two different matches V and W for the same query Q, and suppose that the same ground equations from Q have been tried for V and for W. Then we can compare the degree of success of these matches on an operation by operation basis. If for each operation symbol f we have $\mu_{(V,I_V)}(f) \geq \mu_{(W,I_W)}(f)$, then V is an altogether better match than W. The ideal situation is of course when we find a match that is better than every other match in exactly this sense. However, such an absolutely better match may not exist in the library and we may only get matches that are *maximal* in their degree of success, that is, no other

match is better than them for all functions. This can happen when a query is large enough that two or more parts of the required functionality are available, but their combination is not. In such a case we will find several maximal signature matches that are each best for some fragment of the functionality, but are incomparable among themselves under the μ ordering. An appropriate environment could use this information to help the user synthesize an optimal combined component out of actual components whose corresponding signature matches have maximal μ-measures.

The second measure *SemanticRank* is cruder. It is obtained from the first by assigning to each pair (V, I_V) a real number defined by the equation

$$SemanticRank(V, I_V) = \sum_{f \in \Omega} \mu_{(V,I_V)}(f) ,$$

where Ω is the signature of the query Q.

Example 13. Consider the two matches V_{12} and V_{13} of the component specified by `LIST-OF-ID` for the query `SET-OF-ID` with ground equations as given in Examples 19 and 21 (in Appendix B). The the syntactic and semantic information in $I_{V_{12}}$ and $I_{V_{13}}$ after these equations have been translated and checked is summarized as follows:

For V_{12}:

- `null` goes to `nil`;
- `S + I` goes to `I * S`;
- `S - I` has no translation;
- `I in S` goes to `I in S` and there are 8 ground equations for `_in_`, all of them successful after translation.

For V_{13}:

- `null` goes to `nil`;
- `S + I` has no translation;
- `S - I` goes to `I * S`, and there are 8 ground equations for `_-_`; all of them fail: 4 fail to translate, and 4 fail to be satisfied after translation;
- `I in S` goes to `I in S` and there are 8 ground equations for `_in_`; 4 succeed after translation and 4 fail to translate.

Therefore, we have:

- $\mu_{(V_{12},I_{V_{12}})}(null) = 1$;
- $\mu_{(V_{12},I_{V_{12}})}(+) = 1$;
- $\mu_{(V_{12},I_{V_{12}})}(-) = 0$;

- $\mu_{(V_{12},I_{V_{12}})}(\texttt{in}) = 2$.

And similarly:

- $\mu_{(V_{13},I_{V_{13}})}(\texttt{null}) = 1$;
- $\mu_{(V_{13},I_{V_{13}})}(+) = 0$;
- $\mu_{(V_{13},I_{V_{13}})}(-) = 0$;
- $\mu_{(V_{13},I_{V_{13}})}(\texttt{in}) = 1$.

Therefore V_{12} is an altogether better match than V_{13} according to this measure, and of course also according to the cruder measure *SemanticRank*. This is because we have $SemanticRank(V_{12}, I_{V_{12}}) = 4$ and $SemanticRank(V_{13}, I_{V_{13}}) = 2$.

Given the *SemanticRank* of a component, we can compute its overall *ComponentRank* as follows:

$$ComponentRank = KeywordRank * ProfileRank * SignatureRank * SemanticRank\ .$$

6. An Example

Let us assume that the software base consists of components for generic stack, generic queue, generic bag, and list of natural numbers. Let us also suppose that keywords have been assigned as follows:

Kw(Stack) = {Booch, Data-Structure, Stack},
Kw(Queue) = {Booch, Data-Structure, Queue},
Kw(Bag) = {Booch, Data-Structure, Bag},
Kw(List) = {Booch, Data-Structure, List}.

Now suppose that a user submits a query containing the following information:

1. The keywords are: Booch, Data-Structure, Stack.
2. The partial specification is:

```
Package STACK-OF-NAT is Type Stack;
  function Empty(Out: Stack);
  function Top(In: Stack; Out: Nat);
  function Push(In: Nat, Stack; Out: Stack);
  function Pop(In: Stack; Out: Stack);
  -- Test case 1:
  case 1: top push(1,empty) = top pop(push(6,push(1,empty)));
  -- Test case 2:
  case 2: pop push(7,push(1,empty)) = push(1,empty);
end of Package STACK-OF-NAT;
```

Table 3. Profiles of operations in the query.

Operation	Profile
Empty	110
Top	220
Push	3211
Pop	2201

Table 4. The candidate table.

KeywordRank * *ProfileRank*	*ComponentID*
1.0	G-Stack
0.66	G-Queue
0.66	N-List
0.33	G-Bag

3. The selection criterion is: Show at most 4 components, of highest rank.

Given this query, a program scans the operations in the query to compute its profile β_Q, which is as shown in Table 3.

Next, the program finds the *KeywordIDs* for the query keywords. Using β_Q as input, the program now executes the *FindCandidateComponent* and *DFSFW* algorithms to search for blocks that intersect with β_Q. For each such block, the program indexes to an element in the *ComponentLookupTable* to find the corresponding keywords and *ComponentId*. Using this information, the program can calculate *ProfileRank* and *KeywordRank*, the latter using the equations in Section 4.1. The products of these ranks are sorted and stored with the *ComponentId* in the *CandidateTable*. An example *CandidateTable* is shown in Table 4.

Now we do signature matching, for the query signature against the signatures of G-Queue, G-Stack, N-List, and G-Bag, using the order shown in the table above and the *SignatureMatch* algorithm of Section 4.4. The following maps are obtained:

Query vs. **Stack:**
$V_1 : Q.S \rightarrow G\text{-}Stack.S' = \{$ Stack $\rightarrow$ Stack, Nat $\rightarrow$ Elt $\}$
$V_1 : Q.\Sigma \rightarrow G\text{-}Stack.\Sigma' = \{$ top $\rightarrow$ top, push $\rightarrow$ push, pop $\rightarrow$ pop, empty $\rightarrow$ create $\}$
Query vs. **Queue:**
$V_1 : Q.S \rightarrow G\text{-}Queue.S' = \{$ Stack $\rightarrow$ Queue, Nat $\rightarrow$ Elt $\}$
$V_1 : Q.\Sigma \rightarrow G\text{-}Queue.\Sigma' = \{$ pop $\rightarrow$ pop, top $\rightarrow$ front, push $\rightarrow$ add, empty $\rightarrow$ empty $\}$
Query vs **N-List**:
$V_1 : Q.S \rightarrow N\text{-}List.S' = \{$ Stack $\rightarrow$ List $\}$
$V_1 : Q.\Sigma \rightarrow N\text{-}List.\Sigma' = \{$ top $\rightarrow$ car, push $\rightarrow$ cons, pop $\rightarrow$ cdr, empty $\rightarrow$ nil $\}$
Query vs. **G-Bag:**
$V_1 : Q.S \rightarrow G\text{-}Bag.S' = \{$ Stack $\rightarrow$ Bag, Nat $\rightarrow$ Elt $\}$
$V_1 : Q.\Sigma \rightarrow G\text{-}Bag.\Sigma' = \{$ push $\rightarrow$ add, empty $\rightarrow$ nil $\}$

Table 5. Signature rank table for candidate components.

Map	*SignatureRank*	*ComponentID*
1	1.0	G-Stack
1	1.0	G-Queue
1	1.0	N-List
1	0.5	G-Bag
2	0.5	G-Bag

$V_2 : Q.S \rightarrow G\text{-}Bag.S' = \{$ Stack $\rightarrow$ Bag, Nat $\rightarrow$ Elt $\}$
$V_2 : Q.\Sigma \rightarrow G\text{-}Bag.\Sigma' = \{$ push $\rightarrow$ delete, empty $\rightarrow$ nil $\}$

Once the signature maps have been computed, we can calculate the signature rank for the components. The results of ranking are sorted and stored in the signature rank table, as shown in Table 5.

Once the signatures ranks have been computed, the following steps are performed:

1. Translate the query test cases by applying V:

   (A) **Query** vs. **Stack**:

   top push(1,empty) = top pop(push(6,push(1,empty))) →
   top push(1,create) = top pop(push(6,push(1,create)));

   pop push(7,push(1,empty)) = push(1,empty) →
   pop push(7,push(1,create)) = push(1,create).

   (B) **Query** vs. **N-List:**

   top push(1,empty) = top pop(push(6,push(1,empty)))
   → car cons(1,nil) = car cdr(cons(6,cons(1,nil)));

   pop push(7,push(1,empty)) = push(1,empty) →
   cdr cons(7,cons(1,nil)) = cons(1,nil).

   (C) **Query** vs. **Queue:**

   top push(1,empty) = top pop(push(6,push(1,empty))) →
   front(add(1,empty)) = front(pop(add(6,add(1,empty))));

   pop push(7,push(1,empty)) = push(1,empty) →
   pop(add(7,add(1,empty))) = add(1,empty).

   Note that the test cases for G-Bag cannot be translated because the top and pop operations cannot be translated. Therefore, G-Bag has semantic rank 0.

2. Instantiate the formal parameter(s) with actual parameter(s) using the OBJ3 `make` command. This yields an non-generic component. This step is used on G-Queue and G-Stack, but not N-list.

3. Append the translated test cases to the end of the G-Queue, G-Stack, and N-List code, with OBJ3's reduce command. These transformations are shown below for Nat-Stack (G-Stack instantiated with Nat):

```
obj GENERIC-STACK[X :: TRIV] is sort Stack .
  protecting NAT .
  op create : -> Stack .
  op isempty : Stack -> Bool .
  op push : Elt Stack -> Stack .
  op pop : Stack -> Stack .
  op top : Stack -> Elt .
  var S : Stack .
  var X : Elt .
  eq top(push(X,S)) = X .
  eq pop(push(X,S)) = S .
  eq isempty(S) = if S == create then true else false fi .
endo

*** This is an instantiation statement
make NAT-STACK is GENERIC-STACK[NAT] endm

*** Here are the translated test cases with reduce
*** commands
reduce top(push(1,create)) == top(pop(push(6,push(1,create))))
reduce pop(push(7,push(1,create))) == push(1,create) .
```

Next, this specification is given to OBJ3 for execution. The result is shown below:

```
suns7-caps >> obj3
         OBJ3 version 2.02 built: 1995 Jan 25 Wed 1:31:24
            Copyright 1988,1989,1991 SRI International
                   1995 Apr 10 Mon 4:19:16
OBJ> in stack
==========================================
obj GENERIC-STACK
==========================================
make NAT-STACK
==========================================
reduce in NAT-STACK : top(push(1,create)) ==
top(pop(push(6,push(1, create))))
rewrites: 4
```

Table 6. Semantic ranks of candidate components.

Map	*SemanticRank*	*ComponentID*
1	4.0	G-Stack
1	4.0	N-List
1	0.0	G-Queue
1	0.0	G-Bag
2	0.0	G-Bag

Table 7. Output table showing the ranks of components.

Rank	*SemanticRank*	*ComponentID*
1	4.0	G-Stack
2	2.64	N-List
3	0.0	G-Queue
3	0.0	G-Bag

```
result Bool: true
==========================================
reduce in NAT-STACK : pop(push(7,push(1,create))) == push(1,create)
rewrites: 2
result Bool: true
OBJ> quit
Bye.
```

Now we can compute the semantic ranks for all components, as shown after sorting in Table 6. We see that there are two full matches here, for G-Stack, since the test cases in the query are consistent with the behavior of Nat-Stack, and for N-List. The latter match suggests that a stack can be implemented with a list. Once the semantic ranks are found, we can compute the final *ComponentRank*, using the *ComponentID*s as pointers to the *SignatureRank* and *CandidateTable*, and computing their product. These values are again sorted, producing the results shown in the *OutputTable* in Table 7.

We see that there is now only one component that fully meets the query, namely, G-Stack. N-List ranks second, since it passed through the profile, signature, and semantic filters. G-Bag was eliminated by the semantic filter because two of its operations could not be resolved. G-Queue also ranks third through its failure in semantic filtering.

7. Conclusions and Future Work

This paper proposes a software architecture and method for automated retrieval of reusable software components. The architecture was designed with the goal of improving the precision and recall of search while preserving ease of use. The main ideas are: (1) rank

candidate components by how well they fit the query; (2) use multi-level filtering to make search efficient, where early stages rank components by syntactic criteria; and (3) use test cases (represented as ground equations) for queries. Additional ideas are to organize the search incrementally, and to use profiles to simplify signature matching. Our hypothesis is that this gives a practically useful method combining the advantages of keyword search, which is easy to use but not very discriminating, and specification-based search, which is potentially very accurate but difficult to use. This paper has presented some preliminary examples to illustrate the approach and show that it works on a small scale. Experiments to test our hypothesis about the effectiveness of this method are underway [26]. The rest of this section describes our plan for experiments to test the hypothesis, points out some areas where future research is needed, and discusses some wider implications of our results.

7.1. Experimentation

The practical effectiveness of methods for software component search can only be tested with experiments in realistic settings. Thus, the ideas proposed here should be implemented and tested on a large sample of queries against real software libraries, such as the Booch library, and CAPS software bases for various concrete application areas. Separate aspects of our proposal should be tested separately and compared. In particular, the use of just ground queries should be compared with the use of more sophisticated queries. Various choices for the stages of filtering should be compared; e.g., that illustrated in Figure 2 should be compared with others based on the two stages of semantic filtering suggested in Appendix Sections B.1 and B.2. This will require organizing the testbed implementation in a flexible way, so that various combinations of features can easily be tried separately. Statistics should be compiled to determine the effectiveness and efficiency of each variant.

Experiments of this kind are currently being planned in connection with the CAPS project, using the specification and term rewriting capabilities of OBJ3 to retrieve modules from CAPS software bases [26].

It would also be useful to compile statistics on differences in recall and precision among the various methods discussed in this paper, including keyword search and pure specification-based search. This should be done using a combination of software libraries covering a range of applications. The result should test our conjectures about the limitations of keyword search.

Another issue for experimentation is the most efficient way to evaluate test cases in queries. The performance of term rewriting for executing query test cases in component specifications should be compared with executing the test cases directly in compiled code for the component implementations. Since the number of test cases per query is likely to be small and since new code will have to be generated, compiled and loaded to invoke the components chosen by semantic matching, this overhead could overwhelm the speed advantage of compiled code over term rewriting, especially since one call to OBJ3 could handle a batch of several specifications with their reductions. The costs involved here should be checked experimentally. The difference between these approaches should be measured to help decide if it is worth the effort of associating formal specifications with each component in the software base.

7.2. Future Research and Implications

Concrete heuristics and guidelines to help users formulate queries would be useful. If the guidelines can be stated precisely, then it should be possible to compare their effectiveness experimentally, and to provide at least partial automation for generating sets of test cases. One possibility is to state guidelines as rule sets for constructing the left sides of the ground equations in a test set, corresponding to the input data part of the test cases; users would then provide the corresponding outputs.

The choice of rank functions is another area for further study. Alternatives, such as attaching weights reflecting relative importance of different test cases, should be evaluated experimentally. The proper weighting of the importance of the number of test cases that match for a given operation relative to the number of operations that are supported by a given number of test cases should also be further explored. Using the relation $\equiv$ in Section 4.4 means that some operation matches will be better than others; the definition of rank for signature matching could be modified to take account of this, and then it would be interesting to see if this helps with retrieval.

The PSDL language is an attractive candidate for expressing queries because it is already being used for design representation in CAPS. PSDL provides a fixed syntax for keywords and signatures, and an open syntax for formal specifications, where OBJ3 code could be put.

In addition to OBJ, it would be useful to explore using other languages to represent the formal specifications. For example, it would be interesting to see if the techniques suggested here would work for FOOPS, which is an extension of OBJ that handles states, and for Eqlog, which extends OBJ with features of logic programming. It would also be interesting to assess the value of Eqlog for matching, and to explore the tradeoffs between expressive power and computational requirements. Eqlog has been implemented by Diaconescu [1] as extension of the OBJ3 interpreter, so there would be definite performance limitations.

Some other issues that could be further explored include the following: theoretical aspects of matching generic and non-generic queries to generic modules, by instantiating the parameters; the treatment of subsorts (e.g., is there a finer equivalence on sorts for use in profiles and signature matching than the relation $\equiv$ introduced in Section 4.3; user interface and algorithmic aspects of non-ground queries; the use of behavioral satisfaction; matching generic queries; and further techniques for eliminating modules that cannot match quickly.

Finally, the authors feel that the ideas in this paper may have implications for software testing. In particular, the techniques for generating test cases to be used in matching queries could also be used to test the correctness of code that is supposed to implement a module. It would be interesting to compare this with standard testing techniques.

Appendices

A. Algebraic Specification

This appendix does not attempt to give a complete exposition of algebraic specification; instead, for the sake of completeness, it sketches some definitions and results that are needed in the body of the paper.

We may use the symbol ";" to denote the composition of maps, and [] for the empty string. Sometimes we let "iff" abbreviate "if and only if".

Our first concept is a **signature** (S, Σ), which consists of a **sort set** S with elements called **sorts** (or **types**), plus a family Σ of sets $\Sigma_{w,s}$ of **function** (or **operation**) **symbols**, where $f \in \Sigma_{w,s}$ has **argument sorts** $w = s_1 \ldots s_n$ (for $s_i \in S$ and $n \geq 0$) and has **value sort** $s \in S$; we may write $f\colon s_1 \ldots s_n \to s$ in this case. We also call w the **arity** of f, and w, s the **rank** of f. When $n = 0$, we have $f \in \Sigma_{[],s}$ is a **constant** of sort s, where [] denotes the empty string of input sorts.

We also abbreviate (S, Σ) by just Σ. Notice that, following the tradition of ADJ [15], this notion of signature allows *overloading*, in the sense that the same symbol f can occur in more than one of the sets $\Sigma_{w,s}$. A **subsignature** of a signature (S, Σ) is a signature (S', Σ') such that $S' \subseteq S$ and $\Sigma'_{w,s} \subseteq \Sigma_{w,s}$ for each w and s over S'.

Everything that we say here extends from the many-sorted case to the order-sorted case, by adding a partial ordering relation to the sort set S and making some additional assumptions, as discussed in [8] and [13]. However, we do not discuss the details here.

A Σ-**term** t is an expression composed from the operations in Σ and possibly some variable symbols in a way that respects their sorts; a Σ-term is a **ground** term if it contains no variable symbols. A Σ-**equation** consists of a pair of Σ-terms, t and t', preceded by a declaration of any variable symbols that they use, written in the form $(\forall X)\ t = t'$ where X is the set of declarations; this may be abbreviated $t = t'$. In a **ground equation**, both terms t and t' are ground terms. An **equational theory**, also called an **algebraic specification**, consists of a signature Σ and a set E of Σ-equations, written $T = (S, \Sigma, E)$.

A **signature map** from a signature (S, Σ) to a signature (S', Σ') consists of a sort map $V\colon S \to S'$ and an operation map $V_{w,s}\colon \Sigma_{w,s} \to \Sigma'_{V(w),V(s)}$ for each string w of sorts from S and each sort $s \in S$. If t is a Σ-term, then $V(t)$ denotes its translation under V, the result of substituting $V(f)$ for each function symbol f in t. Similarly, $V(e)$ denotes the translation of an equation e.

A **specification map** from (S, Σ, E) to (S', Σ', E') is a signature map $V\colon (S, \Sigma, E) \to (S', \Sigma', E')$ such that for each $e \in E$, $V(e)$ is a provable consequence of E'.

A **constructor signature** for $T = (S, \Sigma, E)$ is a subsignature Δ of Σ such that every ground Σ-term is provably equal to a ground Δ-term using the equations in E; the elements of Δ are called **constructors**.

The **loose** semantics of a specification $T = (S, \Sigma, E)$ is the class of all Σ-algebras that satisfy E, while the **initial** semantics of T is the class of all initial Σ-algebras that satisfy E. The latter includes the Σ-term algebra, denoted T_Σ, whose elements are the Σ-terms when E is empty. When E is non-empty, the corresponding concrete Σ-algebra is the quotient of the term Σ-algebra by the ground equations that are consequences of E; this algebra is denoted T_E. Any other initial (S, Σ, E)-algebra is isomorphic to this one.

A **persistent** specification map from (S, Σ, E) to (S', Σ', E') has the property that each model of the first specification expands freely to a model of the second specification. (See [12] for the precise technical definition.) A **parameterized specification** is a specification map $V\colon (S, \Sigma, E) \to (S', \Sigma', E')$, which is persistent, and an inclusion, i.e., which has $S \subseteq S'$, $\Sigma \subseteq \Sigma'$ and $E \subseteq E'$.

Given a set X of variable symbols, a Σ-**substitution** is a (sorted) map $\theta\colon X \to T_\Sigma$

assigning a Σ-term to each variable symbol. If t is a Σ-term, then $\theta(t)$ denotes the result of substituting $\theta(x)$ for each variable symbol x occurring in t.

A Σ-**rewrite rule** is a Σ-equation $(\forall X)\ t = t'$ such that each variable occurring in t' also occurs in t. Now let t_1 be a term with a subterm t_0 such that there is a substitution θ such that $\theta(t) = t_0$. Let t_2 be t_1 with $\theta(t')$ substituted for the subterm t_0. Then we say t_1 **rewrites** to t_2 using $(\forall X)\ t = t'$. If t_1 rewrites to t_2 using some equation in E, we write $t_1 \Rightarrow_E t_2$ or just $t_1 \Rightarrow t_2$. Let $\stackrel{*}{\Rightarrow}$ be the transitive closure of $\Rightarrow$; it is the **rewriting relation** defined by E. A Σ-**term rewriting system** (Σ-**TRS**) is a set of Σ-rewrite rules. A Σ-TRS is **Church-Rosser** if whenever $t_0 \stackrel{*}{\Rightarrow} t_1$ and $t_0 \stackrel{*}{\Rightarrow} t_2$ then there is a Σ-term t_3 such that $t_1 \stackrel{*}{\Rightarrow} t_3$ and $t_2 \stackrel{*}{\Rightarrow} t_3$. A Σ-TRS is **terminating** if there is no infinite sequence $t_1 \Rightarrow t_2 \Rightarrow t_3 \Rightarrow \ldots$ of rewrites. A Σ-TRS is **canonical** if it is Church-Rosser and terminating. A Σ-term t is called **irreducible** or **reduced** if there is no t' such that $t \Rightarrow t'$. If $t \stackrel{*}{\Rightarrow} t'$ and t' is reduced, then t' is called a **normal form** of t. The process of computing a reduced form of a term is called **reduction**.

The following result is basic for this paper:

THEOREM 14 *A ground equation $t = t'$ is provable from a canonical TRS E iff the normal forms of t and t' under E are (syntactically) equal.*

Several examples and certain remarks in this paper assume some familiarity with OBJ, for details of which see [6, 10, 16]. OBJ supports both loose and initial specifications based on order sorted (i.e., subtyped) algebra; it has a powerful generic module system, and it supports execution through term rewriting. It is useful to note that algebraic specifications for software modules are essentially always canonical in practice; this is shown by experience with a great many specifications written in OBJ.

B. Advanced Topics in Semantic Matching

The following subsections discuss some more advanced topics in semantic matching, under the assumption that there are no subsorts. Most of this material is based on [14]. The first subsection explains how to generate a set of equations using *patterns* to represent the data sets that can be inputs to a given function. If we succeed in proving the ground equations for all such instances, then our confidence is increased that the candidate function does behave as desired. The second subsection explains how to generate a set of equations consisting of ground instances of the equations in Q. After that, we discuss the more sophisticated notion of behavioral matching, which captures the "black box" behavior of a component, ignoring how that behavior is actually realized. The final subsection discusses the total semantic verification of components, as opposed to using partial semantic verification for retrieval.

B.1. Generating and Checking Ground Equations from Patterns

As before, we assume that all specifications for components in the library are Church-Rosser and terminating. In addition, for the procedure of this subsection, we also assume

that the query Q is Church-Rosser and terminating. Moreover, we assume that for the query specification Q a subsignature of constructors generating all the irreducible terms is given. This means that it can be proved that for each ground term t in the signature of the query specification, there is an irreducible ground term t' in the constructor subsignature such that the equation $t = t'$ is a logical consequence of the equations in the given specification. Because the equations are Church-Rosser and terminating, t' can be easily computed using the equations as rules to rewrite t to an irreducible term t'.

Example 15. The following query for a "set of identifiers" module is Church-Rosser and terminating, and has a subsignature of constructors generating the irreducible terms. We use OBJ3 notation; note that the modules `ID` and `BOOL`, of identifiers and of Booleans, are imported by this module.

```
obj SET-OF-ID is protecting ID BOOL .
  sort Set .
  op null : -> Set .
  op _+_ : Set Id -> Set .
  op _-_ : Set Id -> Set .
  op _in_ : Id Set -> Bool .
  vars I I' : Id .
  var S : Set .
  eq null - I = null .
  eq (S + I) - I' = if I == I' then S - I' else (S - I') + I fi
  eq I in null = false .
  eq I in (S + I') = if I == I' then true else I in S fi
endo
```

Note that the subsignature of constructors is determined by the identifiers and Booleans viewed as constants, plus `null` and the operation `_+_`.

To achieve the semantic validation we have in mind, we begin with ground equations of the form $t = t'$, where t' is an irreducible constructor term that is generated automatically by rewriting t with the equations in Q. To select representative ground terms t, we use the following:

Definition 16. Given a signature Ω and a subsignature of constructors Δ, a **pattern term** is an Ω-term of the form $f(u_1, \ldots, u_n)$ with no repeated variables, where $f\colon s_1 \ldots s_n \to s$ is a function symbol in $\Omega - \Delta$ and where the u_i are Δ-terms of the form $g_i(x_1, \ldots, x_{m_i})$ for some constructor symbol $g_i \in \Delta$ and variables x_i; this includes the case of constants where $m_i = 0$. By convention, we identify two pattern terms if one can be obtained from the other through a bijective renaming of variables.

Example 17. For Ω the signature in Example 15 and Δ its subsignature of constructors, we get the following pattern terms:

```
I in null
```

```
I in (S + I')
null - I
(S + I') - I
```

To generate representative ground terms for each pattern term, we use a *random term generator* that chooses for each variable in a term pattern a constructor term of the appropriate sort. Such a random term generator can be biased towards choosing terms of relatively small depth, which will usually require less time for reduction to normal form.

Example 18. A random term generator might generate the following set R of representative ground terms, where for each pattern two terms are generated:

```
b in null
a in null
b in (null + a)
a in (null + a + b)
null - b
null - a
(null + a) - b
(null + b + a) - a
```

Since we assume the equations in Q are Church-Rosser and terminating, each such term t can be rewritten into an irreducible constructor term t' using the equations in Q. We then consider the set of all equations $t = t'$ of this form for $t \in R$.

Example 19. For the SET-OF-ID query in Example 15 and the set of representative terms R just described, we get the following set of ground equations:

```
b in null = false
a in null = false
b in (null + a) = false
a in (null + a + b) = true
null - b = null
null - a = null
(null + a) - b = (null + a)
(null + b + a) - a = (null + b)
```

We know from our earlier discussions that, for a signature match V: $Q \to T_M$ such that the translation $V(t) = V(t')$ is defined, the equation $V(t) = V(t')$ must be provable from T_M in order for V to be correct, regardless of whether Q and T_M are interpreted initially or loosely. Since we assume that the component specifications T_M are all Church-Rosser and terminating, the provability of $V(t) = V(t')$ from T_M can be settled automatically by comparing the irreducible forms of $V(t)$ and $V(t')$ under rewriting by the equations in T_M. This gives us a necessary condition that can be checked for each candidate signature match.

Example 20. For the SET-OF-ID query in Example 15, the library may contain among other things the list of identifiers module in Example 1, whose full algebraic specification is given by:

```
obj LIST-OF-ID is protecting ID BOOL .
  sort List .
  op nil : -> List .
  op _*_ : Id List -> List .
  op _in_ : Id List -> Bool .
  vars I I' : Id .
  var L : List .
  eq I in nil = false .
  eq I in (I' * L) = if (I == I') then true else I in L fi .
endo
```

Each of the matches V_1–V_{13} then induces a partial translation of the equations in Example 19. For example, V_{12} yields the partial translation

```
b in nil = false
a in nil = false
b in (a * nil) = false
a in (b * a * nil) = true
```

When the left and right sides of each equation are evaluated in the LIST-OF-ID module, we get the following identities:

```
false = false
false = false
false = false
true = true
```

Therefore the LIST-OF-ID module satisfies all the translated ground equations under the match V_{12}. By contrast, the match V_{13} yields the partial translation

```
b in nil = false
a in nil = false
b * nil = nil
a * nil = nil
```

and when the left and right sides of each equation are evaluated in the LIST-OF-ID module we get the equations

```
false = false
false = false
b * nil = nil
a * nil = nil
```

where the first two identities show that the corresponding equations are satisfied under V_{13}, whereas the last two equations, having both sides in canonical form and different, show that those translated equations are *not* satisfied by the LIST-OF-ID module under V_{13}.

B.2. Generating and Checking Ground Equations from the Query

The idea of randomly generating ground instances can also be applied to the equations in the query Q. Thus for each variable x in an equation $t = t'$ in Q, we can randomly choose a constructor term $\theta(x)$ of the same sort to replace that variable and thus obtain a ground instance $\theta(t) = \theta(t')$ of the equation, where θ is the substitution. By the general principles already discussed, it follows that a *necessary* condition for the (partial, since V may not be totally defined) correctness of a signature match V: $Q \to T_M$, regardless of whether Q and T_M are initial or loose, is that the translation $V(\theta(t)) = V(\theta(t'))$ is provable from T_M. Because T_M is Church-Rosser and terminating, we can settle the issue of whether $V(\theta(t)) = V(\theta(t'))$ is provable from T_M by comparing the irreducible forms of $V(\theta(t))$ and $V(\theta(t'))$ under rewriting by the equations in T_M.

We can use ground equations generated from the equations in Q in the way just explained to serve as a second semantic filter to select the best answers to the query Q, after having checked the equations generated from patterns as our first semantic filter. It is attractive to select matches in stages, because ordering the matches by their semantic correctness (as explained in Section 5) lets us further narrow down the choice set at each stage, which can substantially reduce the amount of useless computation.

Example 21. For the SET-OF-ID query in Example 15, a random term generator might give us the following set of ground equations:

```
null - a = null
null - b = null
(null + a) - b = if a == b then null - b else (null - b) + a fi
(null + a + b) - a = if b == a then (null + a) - a  else
                                          ((null + a) - a) + b fi
a in null = false
b in null = false
b in (null + a) = if b == a then true else b in null fi
a in (null + a + b) = if a == b then true else a in (null + a) fi
```

The partial translation of these ground equations by the match V_{12} for the LIST-OF-ID specification in Example 20 yields the following ground equations:

```
a in nil = false
b in nil = false
b in (a * nil) = if b == a then true else b in nil fi
a in (b * a * nil) = if a == b then true else a in (a * nil) fi
```

When the left and right sides of each equation are evaluated in the LIST-OF-ID module, we get the following identities:

```
false = false
false = false
false = false
true = true
```

Therefore, the LIST-OF-ID module satisfies all the translated ground equations under the match V_{12}. By contrast, the match V_{13} yields the partial translation

```
a * nil = nil
b * nil = nil
a in nil = false
b in nil = false
```

and when the left and right sides of each equation are evaluated in the LIST-OF-ID module we get the equations

```
a * nil = nil
b * nil = nil
false = false
false = false
```

where the last two identities show that the corresponding equations are satisfied under V_{13}, whereas the first two equations, having both sides in canonical form and different, show that those translated equations are *not* satisfied by the LIST-OF-ID module under V_{13}.

As with ground equations generated from patterns, Section 5.1 shows how information about the extent to which these checks succeed gives a criterion for ordering the signature matches.

B.3. Behavioral Satisfaction

To be fully successful in finding the desired components, the semantic approach described so far needs an additional, easy extension. The problem is that, without an explicit notion of what is observable, it is possible for algebraic specifications to be *too concrete* in the sense that they overspecify the module when in reality for all observable purposes other specifications not satisfying the given specification will actually be semantically correct matches.

Example 22. Consider the following variant of the SET-OF-ID specification in which we have added the equation

```
eq (S + I) + I' = (S + I') + I .
```

making the order of the elements in the set irrelevant.

```
obj SET'-OF-ID is protecting ID BOOL .
  sort Set .
  op null : -> Set .
  op _+_ : Set Id -> Set .
  op _-_ : Set Id -> Set .
  op _in_ : Id Set -> Bool .
  vars I I' : Id
  var S : Set
  eq (S + I) + I' = (S + I') + I .
  eq null - I = null .
  eq (S + I) - I' = if I == I' then S - I' else (S - I') + I fi
  eq I in null = false .
  eq I in (S + I') = if I == I' then true else I in S fi .
endo
```

The problem now is that this additional equation is *not* satisfied by our best match, V_{12}, for the `LIST-OF-ID` module. For example, random generation from this equation may yield the ground equation

```
(null + a) + b = (null + b) + a
```

that when translated via V_{12} yields the equation

```
b * (a * nil) = a * (b * nil)
```

where the two terms are already in canonical form and different in the `LIST-OF-ID` module. Therefore, in the account presented so far this will count as a *failure* of the match V_{12}, when in fact it shouldn't.

We are missing information about what should be observable in the query. It is very natural to assume that the internal representation of sets should not be observable, whereas the values inserted in or deleted from the set, as well as answers to the question of whether an element is in a set or not, should be observable. This can be made explicit by declaring the sorts whose data values should not be observable as `hidden`. In the example above we can just declare

```
sort Set [hidden].
```

It turns out that the algebraic theory of data types generalizes in a very natural way to algebraic specifications with hidden sorts. The relevant notion now is not that of satisfaction, but rather that of *behavioral satisfaction*, or *refinement*. Intuitively this means that if two specifications behave the same as far as what can be observed about their behavior from data in the visible sorts, then they are equivalent. A precise definition of this notion as well as several key results were given in [12, 24]. A full development of equational logic with hidden sorts has subsequently been carried out in [9, 11].

A key idea in hidden-sorted equational logic is that when the sort of an equation is hidden, what we care about is not whether the equation itself is satisfied, but only whether all its *observable consequences* are. If this is so, we say that the equation is *behaviorally satisfied* by the model in question. In more detail, what this means is that when we plug the left and right sides of the equation in any context whose sort is visible the resulting equation should be satisfied.

Example 23. For the `SET'-OF-ID` module with the additional declaration that the `Set` sort is `hidden`, the following three equations are behavioral consequences of the equation we have added and should therefore be satisfied by any correct match:

```
I" in (S + I) + I' = I" in (S + I') + I
I" in ((S + I) + I') + J = I" in ((S + I') + I) + J
I" in ((S + I) + I') - J = I" in ((S + I') + I) - J
```

Therefore the following randomly generated ground equations should also be satisfied:

```
a in (null + b) + c = a in (null + c) + b
a in ((null + a) + b) + c = a in ((null + a) + c) + b
a in ((null + b) + c) + d = a in ((null + c) + b) + d
a in ((null + a) + c) + d = a in ((null + c) + a) + d
a in ((null + c) + d) - b = a in ((null + d) + c) - b
a in (((null + b) + c) + d) - b = a in (((null + b) + d) + c) - b
```

After translation via V_{12} we get the following ground equations

```
a in c * (b * nil) = a in b * (c * nil)
a in c * (b * (a * nil)) = a in b * (c * (a * nil))
a in d * (c * (b * nil)) = a in d * (b * (c * nil))
a in d * (c * (a * nil)) = a in d * (a * (c * nil))
```

which when evaluated to canonical form using the equations in `LIST-OF-INT` yield the identities

```
false = false
true = true
false = false
true = true
```

which validate the translated equations.

B.4. Semantic Filtering without Specifications

So far we have assumed that each component M has an algebraic specification T_M that correctly specifies it. Writing such specifications for a large library is a desirable but

nontrivial task. It is therefore worthwhile to ask if the approach developed here can still be useful when algebraic specifications are not available for the components. The answer is positive if some automatic or semiautomatic processing of components can make available the minimal structure that is needed to perform search in the style we have advocated[14]. Specifically, let us assume that the following infrastructure is available:

- There is a function that, given a component M, perhaps enriched with some annotations, produces an algebraic signature (S_M, Σ_M) that describes the functionality of the module M as an abstract data type. We may also need to assume that a subsignature of constructors Δ_M can be extracted from M. This gives the signature part of an algebraic specification T_M of M.

- There is a facility that, given a ground Σ_M-term, can evaluate it in the module M by executing the code of M. We also assume a related facility that, perhaps given certain annotations for M, can test the equality in M of two Σ_M-terms t and t'.

Under these assumptions, given a query $Q = (R, \Omega, E)$ in the usual form, we can proceed as follows:

1. Signature matches $V\colon \Omega \to \Sigma_M$ are computed exactly as before.

2. Random generation of ground instances for patterns and for the equations in Q proceeds exactly as before.

3. Checking the generated equations for a signature match $V\colon \Omega \to \Sigma_M$ is done by translating the terms through V, executing the translated ground terms in the module M, and comparing for equality.

This procedure can give further evidence about the degree of correctness of the answers at the code level, even when we have algebraic specifications for all modules. This is because it can be very costly to verify that a module M actually satisfies its specification T_M. Thus, in addition to checking necessary conditions for a candidate signature match to be (partially) correct at the specification level, we can also *test* that the module M satisfies the semantic requirements imposed by Q in some particular instances, which is the corresponding necessary condition at the code level. Again we can do this in stages, narrowing down the best possible answers to a query by imposing successive filters.

An even simpler, though more limited, version of this approach, which does not require the user to be familiar with algebraic specifications, could begin with a query Q consisting of both a *template* in the programming language in which the components are written, giving the *type structure* of the module searched for, and a *test set*, that is, a collection of inputs for each function together with their expected output values. The facilities assumed for the environment of the library could then automatically translate the template given by the user into a corresponding signature, and the test set into a collection of (ground) equations, thus yielding a query Q in the form of an algebraic specification that can be processed as described above.

B.5. Verifying Equations

The general idea implicit in our approach to search through increasingly stronger filters is that it is worth spending considerable extra effort further checking semantic correctness only for those components for which we already have a high assurance of correctness.

In the end, the successive filters will provide us with a small set of candidate matches that have the highest chance of being semantically correct partial answers to our query. Only on such a small set of answers may it be worthwhile to attempt any formal verification. In fact we can verify two things, namely either the partial correctness of the component's specification under the given match, or the partial correctness of the component's code. Of course the first alternative is less costly but nevertheless it greatly increases our confidence relative to the semantic checks previously performed on randomly chosen ground equations, whereas the second alternative, although being the most satisfactory, is in general much more involved. In fact, following our general idea of only spending substantial extra effort when sufficient assurance is already available, it may be wise to postpone code verification until after the component's specification has itself been proved correct.

For loose specifications, proving the partial correctness of a match V: $Q \rightarrow T_M$ can be done automatically, provided that the equations T_M are Church-Rosser and terminating. All we have to do is translate the equations in Q using V, rewrite their left and right sides to canonical form using the equations in T_M, and check whether the results are identical.

For specifications involving initial semantics, such an automatic procedure is in general not possible, and inductive techniques may be required. However, if the above automatic procedure were to be successful in a case involving initiality requirements, we could at least conclude that the initial algebras for Q_0 and T_M are homomorphically realted via V, where Q_0 is the subspecification of Q determined by the subsignature and equations for which V is defined. Even if we meet only with partial success in such an attempt, in the sense that some translated equations are still not proved identical by rewriting, we can in this way further increase our assurance that the match is correct, and reduce the amount of formal verification that still needs to be done if greater levels of assurance are deemed necessary.

Example 24. Assume that the match V_{12} from `SET-OF-ID` to `LIST-OF-ID` is our best candidate at the end of our search process. Then V_{12} translates the equations

```
I in null = false
I in (S + I') = if I == I' then true else I in S fi
```

into the equations

```
I in nil = false
I in (I' * S) = if (I == I') then true else I in S fi
```

which are, up to renaming of variables, identical to equations in `LIST-OF-ID` and are therefore proved correct by rewriting them to the identities

```
false = false
```

```
if (I == I') then true else I in S fi = if (I == I') then true
                                                else I in S fi
```

Acknowledgements

The research reported in this paper has been supported in part by the Army Research Office. The research of Dr. Meseguer has also been supported in part by the Office of Naval Research, Contract N00014-95-C-0225. We wish to thank Vincent Hong and Tuan Nguyen, who are students at the Naval Postgraduate School, for their valiant help with preparing the manuscript, and Frances Page at Oxford for help with some of the figures.

Notes

1. Such consequences may be either equational consequences or inductive consequences, depending of whether a "loose" or an "initial" semantics is given to the specification T_M (these terms are explained in Appendix A). An advantage of our approach is that it is insensitive to which of these semantics is assumed.
2. Precision and recall are classical terms from information retrieval. Let Q be the set of items that should be returned in answer to the query and let R be the choice set actually returned. Then the **precision** of R is defined to be $|R \cap Q|/|Q|$, while the **recall** of R is $|R \cap Q|/|R|$.
3. This name is from the Latin verb "to hunt".
4. For these authors, the word "signature" refers to the rank of a higher-order function, rather than to the syntactic specification of a software module, as in the algebraic tradition.
5. The Venari project uses the term "partial match" in a more restricted sense than we do; their term corresponding to our "partial match" is "relaxed match".
6. This means that the results of any module expressions inside a module are substituted into the module.
7. These patterns are terms that involve those functions, plus some variables, constants, and constructors, such that all other functional expressions are instances. Definition 16 in Appendix Section B.1 makes this precise.
8. Appendix Section B.4 describes an approach where components do not need associated algebraic specifications.
9. These terms are explained in Appendix A.
10. Note that any set of ground equations with distinct irreducible left sides is automatically Church-Rosser and terminating.
11. The difficulties with keyword search mentioned in Section 2.1 do not apply in this case, because we are only using it as a filter to reduce the search space.
12. We will see how to relax this assumption later.
13. Note that algebraic specification theory and OBJ3 use the word "sort" instead of the word "type." Also, the word "function" is often used instead of "operation." This paper will use the words "sort" and "type" interchangeably, and will also use the words "function" and "operation" interchangeably.
14. When this paper was nearly finished, we discovered that [29] makes similar suggestions.

References

1. R. Diaconescu, "Category-Based Operational Semantics of Equational Logic Programming," PhD thesis, Programming Research Group, Oxford University, 1994.
2. S. J. Dolgoff, "Automated interface for retrieving reusable software components," Master's thesis, Naval Postgraduate School, Monterey, California, 1992.

3. G. Fischer, S. Henninger, and D. Redmiles, "Cognitive tools for locating and comprehending software objects for resue," in *Proceedings, 13th International Conference on Software Engineering*, May 1991.
4. W. B. Frakes and T. P. Pole, "An empirical study of representation methods for reusable software components." *IEEE Transactions on Software Engineering* 20(8), pp. 617–630, August 1994.
5. W. B. Frakes and I. Sadahiro, "Success factors for systematic reuse." *IEEE Software*, pp. 15–19, September 1994.
6. K. Futatsugi, J. Goguen, J.-P. Jouannaud, and J. Meseguer, "Principles of OBJ2," in B. Reid, editor, *Proceedings, Twelfth ACM Symposium on Principles of Programming Languages*. Association for Computing Machinery, 1985, pp. 52–66.
7. J. Goguen, "Suggestions for using and organizing libraries in software development," in S. Kartashev and S. Kartashev, editors, *Proceedings, First International Conference on Supercomputing Systems*, IEEE Computer Society, 1985, pp. 349–360. Also in *Supercomputing Systems*, S. and S. Kartashev, Eds., Elsevier, 1986.
8. J. Goguen and R. Diaconescu, "A short Oxford survey of order sorted algebra." *Bulletin of the European Association for Theoretical Computer Science*, 48, pp. 121–133, October 1992. Guest column in the "Algebraic Specification Column". Also in *Current Trends in Theoretical Computer Science: Essays and Tutorials*. World Scientific, 1993, pp. 209–221.
9. J. Goguen and R. Diaconescu, "Towards an algebraic semantics for the object paradigm," in Hartmut Ehrig and Fernando Orejas, editors, *Proceedings, Tenth Workshop on Abstract Data Types*, Springer, 1994, pp.1–29. *Lecture Notes in Computer Science*, Vol. 785.
10. J. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, and J. Meseguer, "An introduction to OBJ3," in J.-P. Jouannaud and S. Kaplan, editors, *Proceedings, Conference on Conditional Term Rewriting*. Springer, 1988, pp. 258–263. *Lecture Notes in Computer Science*, Vol. 308.
11. J. Goguen and G. Malcolm, "Proof of correctness of object representation," in A. W. Roscoe, ed., *A Classical Mind: Essays in Honour of C.A.R. Hoare*. Prentice-Hall, 1994, pp. 119–142.
12. J. Goguen and J. Meseguer, "Universal realization, persistent interconnection and implementation of abstract modules," in M. Nielsen and E. M. Schmidt, eds., *Proceedings, 9th International Conference on Automata, Languages and Programming*. Springer, 1982, pp. 265–281. *Lecture Notes in Computer Science*, Vol. 140.
13. J. Goguen and J. Meseguer, "Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations." *Theoretical Computer Science* 105(2), pp. 217–273, 1992. Also *Programming Research Group Technical Monograph PRG–80*. Oxford University, December 1989, and Technical Report SRI-CSL-89-10, SRI International, Computer Science Lab, July 1989; originally given as lecture at *Seminar on Types*, Carnegie-Mellon University, June 1983; many draft versions exist, from as early as 1985.
14. J. Goguen and J. Meseguer, "Software component search," Technical report, SRI International, Computer Science Lab, September 1994.
15. J. Goguen, J. Thatcher, and E. Wagner, "An initial algebra approach to the specification, correctness and implementation of abstract data types," Technical Report RC 6487, IBM T. J. Watson Research Center, October 1976. In *Current Trends in Programming Methodology, IV*. Raymond Yeh, ed., Prentice-Hall, 1978, pp. 80–149.
16. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud, "Introducing OBJ," in Joseph Goguen and Grant Malcolm, editors, *Algebraic Specification with OBJ: An Introduction with Case Studies*. Cambridge, to appear. Also Technical Report, SRI International.
17. P. Hall and C. Boldyreff, 1991. *Software Engineer's Reference Book*. Butterworth-Heinemann, 1991.
18. S. Henninger, "Using iterative refinement to find reusable software." *IEEE Software*, pp. 48–59, September 1994.
19. Luqi, "Normalized specifications for identifying reusable software," in *Proceedings of the 1987 Fall Joint Computer Conference*, IEEE, October 1987, pp. 46–49.
20. Luqi, V. Berzins, and R. Yeh, "A prototyping language for real-time software." *IEEE Transactions on Software Engineering* 14(10), pp. 1409–1423, 1988.
21. Luqi and Y. Lee, "Towards automated retrieval of reusable software components," in *Proceedings, AAAI Workshop on Artificial Intelligence and Automated Program Understanding*, July 1992.
22. Luqi and M. Ketabchi, "A computer-aided prototyping system." *IEEE Software*, pp. 66–72, March 1988.
23. Y. Matsumoto, "A software factory: An overall approach to software production," in P. Freeman, ed., *Tutorial on Software Reusability*, pp. 155–178, 1987.
24. J. Meseguer and J. Goguen, "Initiality, induction and computability," in M. Nivat and J. Reynolds, eds.,

Algebraic Methods in Semantics. Cambridge, 1985, pp. 459–541.

25. A. Mili, R. Mili, and R. Mittermeir. "Storing and retrieving software components," in *Proceedings, 16th International Conference on Software Engineering*, 1984, pp. 15–19.
26. D. Nguyen, "Multi-Level Filtering for Software Component Retrieval," PhD. thesis, Naval Postgraduate School, 1995.
27. E. Ostertag, J. Hendler, R. Prieto-Diaz, and C. Braun, "Computing similarity in a reuse library system." *ACM Transaction on Software Engineering and Methodolgy*, pp. 205–228, July 1992.
28. D. Ozdemir, "The design and implementation of a reusable component library and a retrieval/integration system," Master's thesis, Naval Postgraduate School, Monterey, California, 1992.
29. A. Podgurski and L. Pierce, "Retrieving reusable software by sampling behavior." *ACM Transactions on Software Engineering and Methodology* 2(3), pp. 286–303, 1993.
30. R. Prieto-Diaz, "Implementing faceted classification for sofware reuse." *Communication of the ACM*, pp. 89–97, May 1991.
31. E. J. Rollins and J. M. Wing, "Specifications as search keys for software libraries," in *Proceedings of the Eighth International Conference on Logic Programming*, 1991.
32. R. Steigerwald, Luqi, and J. McDowell, "CASE tool for reusable software component storage and retrieval in rapid prototyping." *Information and Software Technology*, pp. 698–705, 1991.
33. R. A. Steigerwald, "Reusable Software Component Retrieval via Normalized Algebraic Specifications," PhD thesis, Naval Postgraduate School, 1991.
34. A. M. Zaremski and J. M. Wing, "Signature matching, a tool for using software libraries," in *Proceedings, ACM Symposium on Foundations of Software Engineering*, 1993. To appear, *Transactions on Software Engineering and Methodology*.
35. A. M. Zaremski and J. M. Wing, "Specification matching of software components," Technical Report CMU-CS-95-127, School of Computer Science, Carnegie-Mellon University, 1995.